Multiplication by an Integer Constant: Lower Bounds on the Code Length

Vincent Lefèvre

INRIA Lorraine, 615 rue du Jardin Botanique, 54602 Villers-lès-Nancy Cedex, France

Abstract

In this paper, we deal with code that performs a multiplication by a given integer constant using elementary operations, such as left shifts (i.e. multiplications by powers of two), additions and subtractions. Generating such a code can also be seen as a method to compress (or more generally encode) integers. We will discuss neither the way of generating code, nor the quality of this compression method, but this idea will here be used to find lower bounds on the code length, i.e. on the number of elementary operations.

Key words: integer multiplication, addition chains, compression

1 Introduction

We consider the problem of generating code to perform a multiplication by an integer constant n, using elementary operations, such as left shifts (i.e. multiplications by powers of two), additions and subtractions. This problem has been studied and algorithms have been proposed in [3,4,7–9,5]. Other operations like right shifts could be taken into account, but we will restrict to the above operations, as this was done in [8,9]. However, the results presented in this paper could straightforwardly be generalized.

In this paper, we are interested in the relation between the multiplication by an integer constant problem and the compression (Section 4). This will allow us to deduce lower bounds on the length of the generated code, also called *program* (Section 5). But we first give a formulation of the problem (the same as in [8,9]) in Section 2 and discuss on bounds on the shift counts in Section 3.

Email address: Vincent.Lefevre@loria.fr (Vincent Lefèvre).
URL: http://www.vinc17.org/research/ (Vincent Lefèvre).

2 Formulation of the Problem

We now formulate the problem. The assumptions and choices we will make for our model are not discussed here. The reader can find more details in [8,9].

An integer x left-shifted c bit positions (i.e. x multiplied by 2^c) is denoted by $x \ll c$, and when we write expressions, the shift has a higher precedence than the addition and the subtraction (contrary to the precedence rules of some languages, like C or Perl). We assume that the computation time of a shift operation does not depend on the value c, commonly called the *shift count*.

In practice, shifts are generally associated with an addition or a subtraction: expressions can be rewritten to *delay* some shift operations. For instance, in $(x \ll 3 + y \ll 8) \ll 1$, instead of performing the shift by 3 immediately, we choose to perform it *after the addition*, by factorizing it: $(x + y \ll 5) \ll 4$. Thus, in our model, the elementary operations will be additions and subtractions where one of the operands is left-shifted by a fixed number of bits (possibly zero), and we assume that these operations take the same computation time, which will be chosen to be a time unit.

Let n be a nonnegative odd integer (this is our constant) [1]. A finite sequence of nonnegative integers u_0, u_1, u_2, ..., u_q is said to be *acceptable* for n if it satisfies the following properties:

- initial value: $u_0 = 1$;
- for all $i > 0$, $u_i = |s_i\, u_j + 2^{c_i}\, u_k|$, with $j < i$, $k < i$, $s_i \in \{-1, 0, 1\}$ and $c_i \geq 0$;
- final value: $u_q = n$.

Thus, from an acceptable sequence for n, we can multiply any arbitrary number x by n, using the same operations as above: the corresponding program iteratively computes $u_i x$ from already computed values $u_j x$ and $u_k x$, to finally obtain nx. Note that the absolute value is used here to make the notations simpler and no absolute values are actually computed: the absolute value means that if $s_i = -1$, the smaller value is subtracted from the larger value.

The problem is to generate, from the number n, an acceptable sequence $(u_i)_{0 \leq i \leq q}$ that is as short as possible; q is called the *quality* or *length* of the corresponding program (it is the computation time when this program is executed under the condition that each instruction takes one time unit).

[1] We choose to restrict to odd integers, because if we want to perform a multiplication by n, we can seek to perform a multiplication by the odd integer of the form $n/2^c$; this choice may lead to longer codes, but the difference is not significant, and this is what algorithms used in practice do, working on odd integers only.

As n is odd, we have $s_i \neq 0$ for all i in a minimal acceptable sequence. This can be shown by delaying the shifts. Indeed, if the delayed shift associated with u_i is denoted δ_i, δ_0 being 0, we can write $d_i = \delta_k + c_i - \delta_j$ and u_i can be replaced by u'_i, such that $u'_0 = 1$ and for $i > 0$:

$$(u'_i, \delta_i) = \begin{cases} (u'_k, \delta_k + c_i) & \text{if } \ s_i = 0, \\ (\left| s_i\, u'_j + 2^{d_i}\, u'_k \right|, \delta_j) & \text{if } \ s_i \neq 0 \ \text{ and } \ d_i \geq 0, \\ (\left| s_i\, u'_k + 2^{-d_i}\, u'_j \right|, \delta_k + c_i) & \text{if } \ s_i \neq 0 \ \text{ and } \ d_i < 0. \end{cases}$$

Then we have $u_i = 2^{\delta_i}\, u'_i$ and $\delta_q = 0$ (since n is odd); therefore the sequence $(u'_i)_{0 \leq i \leq q}$ is acceptable for n. Operations $u'_i = u'_k$ (corresponding to $s_i = 0$) can be removed. Thus, if the sequence (u_i) is minimal, there cannot be such an operation, and $s_i \neq 0$ for all i.

Generating optimal code is very difficult. Therefore one uses heuristics in practice; such heuristics have been described in [3,4,7–9,5]. This paper does not deal with what these heuristics do exactly, but with lower bounds on the length q of any program (either an optimal program or a program generated by some heuristic). The lower bounds will be found by regarding the program as a compressed form of the number n and using results from the theory of information.

3 Bounds on the Shift Count

3.1 Considered Bounds

To obtain an upper bound on the size of an elementary operation (Section 4.3), we need to know a bound on the shift counts c_i. But we currently do not know any proved useful result. We will assume that for any i, c_i is bounded by $S(m)$, where S is a function of the number of bits m of the constant n.

First, we require that $S(m) \geq m$ to be able to compute $2^m - 1$ with one elementary operation. Moreover, currently-known heuristics generate code that always satisfies $c_i \leq m$, leading to $S(m) = m$. But what can we say about the function S when considering the class of optimal programs? To get n with an optimal program, experiments suggest that u_i/n is probably unbounded [2]. However, we do not know any useful result about shift counts. Therefore, we need to consider a conjectured bound, such as $S(m) = \alpha.m$ for some $\alpha \geq 1$ (if the optimal program for the multiplication by n is adequately chosen).

[2] Indeed, two large values u_j and $2^{c_i}\, u_k$ close to each other and computable with few elementary operations may be subtracted to give a much smaller value.

3.2 Some Notations

Consider a program P and its associated shift counts $c_1, c_2, \ldots, c_q$. We denote: $c_{\max}(P) = \max_{1 \le i \le q} c_i$. Let n be a nonnegative odd integer, $q_{\text{opt}}(n)$ the minimal integer such that there is a program that computes n in $q_{\text{opt}}(n)$ elementary operations and $\mathcal{P}(n)$ the set of these minimal (or optimal) programs. We define: $c_{\inf}(n) = \inf_{P \in \mathcal{P}(n)} c_{\max}(P)$ and $c_{\sup}(n) = \sup_{P \in \mathcal{P}(n)} c_{\max}(P)$.

REMARKS: We will prove in Section 3.6 that $c_{\sup}(n)$ is finite for any n (this is not obvious). Moreover, as the program length does not depend on the choice of the optimal program for n, we can choose any optimal program and only $c_{\inf}(n)$ will be useful concerning the lower bounds on the program length: it suffices to require that $S(m) \ge c_{\inf}(n)$ for any m-bit number n; but doing proofs on $c_{\inf}(n)$ is more difficult than on $c_{\sup}(n)$.

3.3 Computed Results

To justify the considered bound $S(m) = \alpha.m$, we performed some computations on small values of n: we considered all the possible acceptable sequences for n satisfying $1 \le n \le 2^{20} - 1$, with the restriction $u_i \le 2^{25}$ for any i (we need some restriction, as the number of acceptable sequences for n is infinite). Of course, the following results are subject to this restriction and should not be regarded as proved results.

We obtained the following result on $c_{\inf}$: For $3 \le m \le 20$, the maximum value of the computed $c_{\inf}(n)$ for all m-bit numbers n is equal to m, and this shift count m is needed only for $n = 2^m - 1$ (computed by $1 << m - 1$).

Concerning $c_{\sup}$, Table 1 gives the maximum value of the computed $c_{\sup}(n)$ for all m-bit numbers n. We stopped at $m = 16$ because the results for $m > 16$ are affected by the chosen restriction $u_i \le 2^{25}$.

m	$\max c_{\sup}$	n
2	2	3
3	3	7
4	4	11
5	5	23
6	7	43

m	$\max c_{\sup}$	n
7	8	91
8	9	187
9	10	379
10	11	683
11	14	1,369

m	$\max c_{\sup}$	n
12	15	2,969
13	17	4,945
14	19	14,709
15	22	22,387
16	25	46,853

Table 1
For $2 \le m \le 16$, the maximum value of the computed $c_{\sup}(n)$ for all m-bit numbers n, and the minimum value of n where this shift count $c_{\sup}(n)$ appears.

For instance, the corresponding optimal code for the 16-bit number $n = 46,853$ is:

$$\begin{aligned}
u_0 &= 1 \\
u_1 &= u_0 \ll 9 \;\; - u_0 = 511 \\
u_2 &= u_1 \ll 2 \;\; - u_0 = 2,043 \\
u_3 &= u_1 \ll 8 \;\; - u_2 = 128,773 \\
u_4 &= u_0 \ll 25 - u_3 = 33,425,659 \\
u_5 &= u_2 \ll 14 - u_4 = 46,853
\end{aligned}$$

We can guess from these results that to prove a bound $S(m) = \alpha.m$, considering only the fact that a program is optimal (as it is sometimes done) would not be sufficient. Thus, the choice of the optimal program is important to avoid very large shift counts like in the above example. This fact is *proved* on a generic example in the following section.

3.4 An Example of Optimal Programs With Large Shift Counts

For $h \geq 2$, consider $n = (1 + 2^h)(1 + 2^{2h})(1 + 2^{4h}) - 2^{7h}$, which is a $6h + 1$-bit number $(2^{6h} + 2^{5h} + 2^{4h} + 2^{3h} + 2^{2h} + 2^h + 1)$ that can be computed with 4 elementary operations only, but using a shift count of $7h$:

$$\begin{aligned}
u_0 &= 1 \\
u_1 &= u_0 \ll h \;\; + u_0 \\
u_2 &= u_1 \ll 2h + u_1 \\
u_3 &= u_2 \ll 4h + u_2 \\
u_4 &= u_3 \qquad\quad - u_0 \ll 7h\,.
\end{aligned}$$

We prove below that this number cannot be computed with fewer than 4 elementary operations, thus $c_{\sup}(n) \geq 7h$ for this number, though there exists a program that computes this number with 4 operations and a maximum shift count of $2h$ (so, not larger than the number size $6h + 1$):

$$\begin{aligned}
u_0 &= 1 \\
u_1 &= u_0 \ll h \;\; + u_0 \\
u_2 &= u_1 \ll h \;\; + u_0 \\
u_3 &= u_2 \ll 2h + u_1 \\
u_4 &= u_3 \ll 2h + u_1\,.
\end{aligned}$$

To prove that $(1+2^h)(1+2^{2h})(1+2^{4h})-2^{7h}$ cannot be computed with fewer than 4 operations, we need the following lemma:

Lemma 1 *Let r be a nonnegative integer, $(s_i)_{1\le i\le r}$ be r integers equal to ± 1 (signs) and $(c_i)_{1\le i\le r}$ be r nonnegative integers. Consider $n=\sum_{i=1}^{r} s_i\,2^{c_i}$. Then there exists a representation of n in binary using the signed digits 0, 1, -1, that has no more than r nonzero digits.*

The lemma can be proved by induction. It is true for $r=0$. Assume that it is true up to $r-1$. Let us prove it for the value r.

If all the c_i's are different, then the representation associated with the above sum is suitable. Otherwise there exist j and k ($j\neq k$) between 1 and r such that $c_j=c_k$. If $s_j\neq s_k$, we can remove the corresponding terms from the sum; this leads to a sum with $r-2$ terms, whose value n can be written with no more than $r-2$ nonzero digits, therefore with no more than r digits. If $s_j=s_k$, we can replace the corresponding terms by a term having the same sign and an exponent equal to c_j+1; this leads to a sum with $r-1$ terms, whose value n can be written with no more than $r-1$ nonzero digits, therefore with no more than r digits. □

Now we can prove the following theorem.

Theorem 2 *For $h\ge 2$, the number $(1+2^h)(1+2^{2h})(1+2^{4h})-2^{7h}=\sum_{i=0}^{6}2^{ih}$ cannot be computed with fewer than 4 elementary operations.*

The number $n=(1+2^h)(1+2^{2h})(1+2^{4h})-2^{7h}$ is represented in binary with 7 digits one, separated by at least a zero (since $h\ge 2$); therefore it cannot be written with fewer digits (well-known result on canonical Booth's recoding). As a consequence of the lemma, when n is expressed as sums and differences of powers of two, there are at least 7 terms in this expression.

With only 1 elementary operation, we have at most 2 terms in the expression. With 2 elementary operations, we have at most 4 terms after expanding the expression. With 3 elementary operations, the values (j,k) associated with i in a program can be, up to an isomorphism of the corresponding DAG:

- $(0,0)$ $(0,0)$ $(1,2)$ $\rightarrow$ 4 terms (after expansion).
- $(0,0)$ $(0,1)$ $(0,2)$ $\rightarrow$ 4 terms (after expansion).
- $(0,0)$ $(0,1)$ $(1,2)$ $\rightarrow$ 5 terms (after expansion).
- $(0,0)$ $(0,1)$ $(2,2)$ $\rightarrow$ 6 terms (after expansion).
- $(0,0)$ $(1,1)$ $(0,2)$ $\rightarrow$ 5 terms (after expansion).
- $(0,0)$ $(1,1)$ $(1,2)$ $\rightarrow$ 6 terms (after expansion).
- $(0,0)$ $(1,1)$ $(2,2)$ $\rightarrow$ 8 terms (after expansion).

Therefore, if n can be computed with fewer than 4 operations, then it *must* be computed with a DAG of the form $(0,0)$ $(1,1)$ $(2,2)$, i.e. we can write: $n = (2^a + s_a)\,(2^b + s_b)\,(2^c + s_c)$ with $a, b, c \geq 1$ and $s_a, s_b, s_c \in \{1, -1\}$.

First, notice that n is congruent to 1 modulo 3:

$$\begin{aligned} n &= (1+2^h)(1+2^{2h})(1+2^{4h}) - 2^{7h} \\ &\equiv (1+2^h) \times 2 \times 2 - 2^h \equiv 1 \pmod 3. \end{aligned}$$

If $a = 1$, then $s_a \neq 1$ (because n is not divisible by 3) and $s_a \neq -1$ (otherwise, the expression can be written with at most 4 terms). Therefore $a \geq 2$. For the same reasons, $b \geq 2$ and $c \geq 2$. As $h \geq 2$, $n \equiv 1 \pmod 4$; therefore $s_a s_b s_c = 1$.

$2^a \not\equiv 0 \pmod 3$, therefore $2^a + s_a \not\equiv s_a \pmod 3$, and as $n \not\equiv 0 \pmod 3$, $2^a + s_a \not\equiv 0 \pmod 3$. The only possibility is: $2^a + s_a \equiv -s_a \pmod 3$. For the same reasons, $2^b + s_b \equiv -s_b \pmod 3$ and $2^c + s_c \equiv -s_c \pmod 3$. As a consequence, $n \equiv (-s_a)(-s_b)(-s_c) = -(s_a s_b s_c) = -1 \pmod 3$, which leads to a contradiction. This proves Theorem 2. □

3.5 Shift Reduction

We now discuss results that could allow us to get proved upper bounds on the shift count (but they still need to be developed).

In Section 3.6, we will prove that $c_{\sup}(n)$ is finite using the following idea. If there are very large shift counts, this means that the binary representations of the values u_i could be split into (at least) two parts separated by a long sequence of zeros and the high-order parts would cancel each other. However, computing these parts would take useless operations, thus increasing the value of q.

Here's an example with $n = 17$ and $q = 3$, where the shift counts $c - 4$ and c can be as large as we want, c being an integer larger or equal to 4:

$$\begin{aligned} u_0 &= 1 \\ u_1 &= u_0 \ll (c-4) + u_0 = 2^{c-4} + 1 \\ u_2 &= u_0 \ll c \quad\;\; - u_0 = 2^c - 1 \\ u_3 &= u_1 \ll 4 \quad\;\; - u_2 = 17 \end{aligned}$$

This can be formalized in the program, in the following way. First, for the sake of simplification, the absolute values are removed: $|u - v|$ is written $u - v$

or $v - u$ depending on whether $u \geq v$ or $u < v$. Then, we work on $\mathbb{Z}[X]$ (polynomials), where each monomial represents a part, the degree-0 monomial being the low-order part. Initially, we have a low-order part only: $u_0 = 1$. The operations remain as before, except when a shift is regarded as "large": in this case, a multiplication by X is performed. For instance, with the previous program and $c = 51$, we can write:

$$\begin{aligned} u_0 &= 1 \\ u_1 &= u_0 X \ll 47 + u_0 \\ u_2 &= u_0 X \ll 51 - u_0 \\ u_3 &= u_1 \ll 4 \quad - u_2 \,. \end{aligned}$$

Thus, $u_3 = 2^4 u_1 - u_2 = 2^4(2^{47}X + 1) - (2^{51}X - 1) = 17$. We notice that the large shifts have yielded cancellations and we obtain a degree-0 monomial. Since the result does not depend on X, we can replace X by 0 (instead of 1) to get a shorter program after its simplification: $u_0 = 1$, $u_1 = u_0 \ll 4 + u_0$.

3.6 Proof of a Large Upper Bound on $S(m)$

We now search for an upper bound on $S(m)$ by using the following idea: If the shift counts can be very large, then, as the number of elementary operations is limited to q, itself bounded above by $\lfloor m/2 \rfloor$ (using Booth's recoding and the naive algorithm as described in [8,9]), there will be at least a "hole" giving two parts in the binary representation of the intermediate results.

Let m be an integer greater than 1, n a m-bit nonnegative odd integer (our constant) and $(u_i)_{0 \leq i \leq q}$ a minimal acceptable sequence associated with n. The shift counts are denoted c_i as in the formulation. Let σ be a permutation of the first q nonnegative integers 1, 2, ..., q that orders the shift counts, $d_i = c_{\sigma(i)}$ and $d_0 = -1$, so that for all $1 \leq j \leq q$, we have $d_j \geq d_{j-1}$. For instance, in the following program

$$\begin{aligned} u_0 &= 1 \\ u_1 &= u_0 \ll 4 + u_0 = 17 \\ u_2 &= u_0 \ll 4 - u_0 = 15 \\ u_3 &= u_2 \ll 7 - u_1 = 1903 \\ u_4 &= u_3 \ll 2 + u_0 = 7613, \end{aligned}$$

we can consider the permutation $\sigma(1) = 4$, $\sigma(2) = 2$, $\sigma(3) = 1$, $\sigma(4) = 3$.

Let j be an integer between 1 and q and let us mark (by multiplying by X) the shifts of lines $\sigma(i)$ for $i \geq j$. For instance, if $j = 3$, this gives:

$$\begin{aligned}
u_0 &= 1 \\
u_1 &= u_0 X << 4 + u_0 = 16X + 1 \\
u_2 &= u_0 << 4 \quad - u_0 = 15 \\
u_3 &= u_2 X << 7 - u_1 = 1919 - 16X \\
u_4 &= u_3 << 2 \quad + u_0 = 7677 - 64X.
\end{aligned}$$

The corresponding polynomial can be written: $A(X).X + b$, where A is a polynomial with integer coefficients and b an integer constant. We have: $A(1) + b = n$.

As q is minimal, we have $A(1) \neq 0$. And as in the computation, each coefficient of X is divisible by 2^{d_j}, then $A(1)$ is divisible by 2^{d_j} and we have: $|A(1)| \geq 2^{d_j}$.

Now, let us find an upper bound on $|b|$. To compute b, we write $X = 0$. A simple reasoning by induction leads to:

$$|b| \leq \prod_{k=1}^{j-1} \left(2^{d_k} + 1\right).$$

We can find an upper bound on $|b|$ by considering upper bounds D_k on d_k. Let us assume that $D_k \geq \max(2^{k-1}, d_k)$ for $1 \leq k \leq j-1$. Then we have:

$$\prod_{k=1}^{j-1} \left(1 + 2^{-D_k}\right) \leq \prod_{k=0}^{j-2} \left(1 + 2^{-2^k}\right) = 2\left(1 - 2^{-2^{j-1}}\right) < 2.$$

Therefore $|b| < 2^{S_j}$ with: $S_j = 1 + \sum_{k=1}^{j-1} D_k$.

As $A(1) + b = n$, we have $|A(1)| \leq n + |b|$. Therefore

$$2^{d_j} \leq |A(1)| \leq n + |b| < 2^m + 2^{S_j}$$

and $d_j \leq \max(m, S_j)$. From this inequality, we can deduce that the smallest shift count is bounded above by m, the next one by $m + 1$, then $2m + 2$, then $4m + 4$, and so on. By induction, we can prove that for $j \geq 2$, the j-th shift count can be bounded by $2^{j-2}(m + 1)$. Note that $m \geq 2^0$ and for $j \geq 2$, $2^{j-2}(m+1) \geq 2^{j-1}$, therefore we lost nothing by requiring that $D_k \geq 2^{k-1}$. As a consequence, we have the following theorem.

Theorem 3 *Let m be an integer greater than 1 and n a m-bit nonnegative odd integer. Consider an optimal program that computes n. The largest shift count is smaller or equal to*

$$\begin{cases} m & \text{if } q = 1 \\ 2^{q-2}(m+1) & \text{if } q \geq 2. \end{cases}$$

As $q \leq \lfloor m/2 \rfloor$, we can take for $m \geq 4$:

$$S(m) = 2^{\lfloor m/2 \rfloor - 2}(m+1)$$

which is asymptotically equivalent to $2^{\lfloor m/2 \rfloor - 2}\, m$. Unfortunately, this upper bound is so large that it will not give us any interesting result in the following.

4 Compression

A program contains nonnegative integers. With the conventional representation of the nonnegative integers in base 2, it is not possible to know where a word representing an integer ends in a sequence of 0's and 1's: we need a prefix code of the integers. Of course, such a code should have a small length complexity (for instance, the well-known base-1 representation 1^n0 is not acceptable). So, we will first describe a method to encode the nonnegative integers efficiently (Section 4.1).

Then we will describe the encoding of an elementary operation and the whole program (Section 4.2), and finally, Section 4.3 will deal with the size of the program.

4.1 Prefix Code of the Nonnegative Integers

The problem of finding a prefix code of the nonnegative integers is related to the *unbounded search problem*, which has been studied by Bentley and Yao [2], Raoult and Vuillemin [10], Knuth [6], and Beigel [1]. However, for the sake of simplicity, we will not choose a prefix code that is as short as the ones defined in [1], in particular because our problem is more general: we need to encode several integers and if we wanted a really short code, taking this fact into account for the choice of the code could be preferable.

The conventional representation of the nonnegative integers in base 2 gives an encoding in $\lfloor \log_2(n) \rfloor + 1$ bits (for $n \geq 1$). So, we look for a prefix

code that would be in slightly more than $\lfloor \log_2(n) \rfloor$ bits, i.e. a complexity in $\log_2(n) + o(\log_2(n))$. The idea is to express the length of the word in an efficient way. We could give it in the base-1 representation, but this is not sufficient to achieve our goal. So, we will give it in base 2 and give the length of the length in base 1 (in fact, a variant of that).

More precisely, 0, 1, 2 and 3 will respectively be encoded by 000, 001, 010, 011. For $n \geq 4$, k denotes the number of bits of n without the first 1, i.e. n has $k+1$ bits ($k \geq 2$). For $k \geq 2$, h denotes the number of bits of k without the first 1, i.e. k has $h+1$ bits ($h \geq 1$). The code word of n will consist of h digits 1, followed by a 0, the h bits of k without the first 1, and the k bits of n without the first 1. For instance, $n = 17$ is 10001 in base 2, and without the first 1, we get the 4 digits 0001; thus $k = 4$, which is 100 in base 2, and without the first 1, we get the 2 digits 00; thus $h = 2$, and 17 is encoded by 110 00 0001. Table 2 gives the code words for a few integers.

integer	code word
0	000
1	001
2	010
3	011
4	10 0 00
5	10 0 01
6	10 0 10
7	10 0 11
8	10 1 000
15	10 1 111

integer	code word
16	110 00 0000
31	110 00 1111
32	110 01 00000
63	110 01 11111
64	110 10 000000
127	110 10 111111
128	110 11 0000000
255	110 11 1111111
256	1110 000 00000000
511	1110 000 11111111

Table 2
Encoding of a few integers.

The length of the code word corresponding to a nonnegative integer n is:

$$C(n) = \begin{cases} 3 & \text{if } n \leq 3, \\ \lfloor \log_2(n) \rfloor + 2 \ \lfloor \log_2(\log_2(n)) \rfloor + 1 & \text{if } n \geq 4. \end{cases}$$

We have: $C(n) = \log_2(n) + o(\log_2(n))$, i.e. $C(n) \sim \log_2(n)$.

4.2 Encoding of the Program

An elementary operation has the form $u_i = |s_i\, u_j + 2^{c_i}\, u_k|$. Thus it suffices to encode s_i and the nonnegative integers c_i, j and k. The four words may be simply concatenated. As s_i can take three possible values (-1, 0 and 1), we

use two bits[3] to encode s_i. The fourth combination of these two bits can be used to indicate the end of the program (*stop word*). The nonnegative integers c_i, j and k are encoded as described in Section 4.1.

The program is encoded by concatenating the code words of its elementary operations, and the two-bit stop word at the end.

4.3 Size of the Program

We are now interested in an upper bound on the size of the encoded program.

First, let us find an upper bound on the size of the i-th elementary operation. The integer c_i is bounded by $S(m)$, as said in Section 3. The integers j and k are bounded by $i-1$. Thus, the i-th elementary operation can be encoded with at most $2 + C(S(m)) + 2\,C(i-1)$ bits and a program of length q, i.e. having q elementary operations, can be encoded with at most

$$\sum_{i=1}^{q} (2 + C(S(m)) + 2\,C(i-1)) + 2 = q\,(2 + C(S(m))) + 2\sum_{i=0}^{q-1} C(i) + 2$$

bits. The main term of $C(i)$ is $\lfloor \log_2(i) \rfloor$ (for $i \geq 4$), so we wish to evaluate the corresponding sum, denoted $L(p)$. We can prove by induction that

$$L(p) = \sum_{i=1}^{p} \lfloor \log_2(i) \rfloor = (p+1)\ \lfloor \log_2(p) \rfloor - 2^{\lfloor \log_2(p) \rfloor + 1} + 2.$$

There is not much difference with $p\ \lfloor \log_2(p) \rfloor$, and in particular, $L(p)$ is asymptotically equivalent to $p\,\log_2(p)$. So, without too much loss, we can bound i by q in the above formula.

Therefore a program of length q can be encoded with at most

$$B(m,q) = q\,(2 + C(S(m)) + 2\,C(q-1)) + 2$$

bits. This bound is asymptotically equivalent to $q\,(\log_2(S(m)) + 2\,\log_2(q))$, and if we assume that $S(m) = \alpha.m$ as in Section 3, we have:

$$B(m,q) \sim q\,(\log_2(m) + 2\,\log_2(q)).$$

[3] This is not necessarily the best choice, in particular knowing the fact that $s_i = 0$ can be avoided (except for the last elementary operation, if even integers are accepted as the input). But as this will not make a significant difference, we prefer to keep the general case.

5 Lower Bounds on the Length of the Program

First, we define the following notation. If f and g are two nonnegative functions on some domain, we write $f(x) \gtrsim g(x)$ if there exists a function ε such that $|\varepsilon(x)| = o(1)$ and $f(x) \geq g(x)\,(1 + \varepsilon(x))$. This is also equivalent to say that there exists a function ε' such that $|\varepsilon'(x)| = o(1)$ and $f(x)\,(1 + \varepsilon'(x)) \geq g(x)$.

For each m-bit nonnegative odd integer n, we consider an optimal program that computes n, adding the following restriction on the shift counts to the formulation: $\forall i,\ c_i \leq S(m)$; of course, if $S(m)$ is large enough, this is no longer a restriction. For each n, the corresponding program length q_n is thus completely defined.

5.1 Worst Case

Let us consider the nonnegative odd integers having exactly m bits in their binary representation. They are 2^{m-2} such integers. As all the corresponding encoded programs must be different, there exists an integer whose code word has at least $m-2$ bits; let us denote by q_{worst} the length of the corresponding program. Therefore, one has $B(m, q_{\text{worst}}) \geq m-2$. This will give a lower bound on the program length q_{worst} in the worst case, as a function of m.

Asymptotically, under the $S(m) = \alpha.m$ assumption, we have, when $m \to \infty$:

$$B(m, q_{\text{worst}}) \sim q_{\text{worst}}\,(\log_2(m) + 2\,\log_2(q_{\text{worst}}))$$

Thanks to experiments, we can guess that $\log_2(q_{\text{worst}}) \sim \log_2(m)$; since $q_{\text{worst}} \leq m$, we can bound $\log_2(q_{\text{worst}})$ by $\log_2(m)$ without too much loss and write: $3\,q_{\text{worst}}\,\log_2(m) \gtrsim B(m, q_{\text{worst}})$. Using $B(m, q_{\text{worst}}) \geq m-2$, we deduce: $3\,q_{\text{worst}}\,\log_2(m) \gtrsim m$. As a consequence, we obtain Theorem 4.

Theorem 4 *Let $m \geq 2$. For any positive odd integer n having exactly m bits in its binary representation, consider an acceptable sequence computing n and let q_n denote its length. Let $q_{\text{worst}} = \max_n q_n$ be the maximum length. Assume that the shift counts are bounded above by a function $S(m) = \alpha.m$ (α being a positive constant). Then*

$$q_{\text{worst}} \gtrsim \frac{m}{3\,\log_2(m)}.$$

Note that this also proves what we have guessed: $\log_2(q_{\text{worst}}) \sim \log_2(m)$.

We can also deduce an exact (instead of asymptotic) bound, for $m \geq 4$:

$$q_{\text{worst}} > \frac{m-4}{2 + C(S(m)) + 2\,C(m)}$$

i.e. q_{worst} is larger than:

$$\frac{m-4}{3\,\log_2(m) + 4\,\lfloor \log_2(\log_2(m)) \rfloor + 2\,\lfloor \log_2(\log_2(\alpha.m)) \rfloor + \log_2(\alpha) + 6}.$$

Of course, this bound can easily be (very slightly) improved. But this is not the goal of this paper.

We do not know if the lower bound given by Theorem 4 is reached. The only currently known upper bound for the worst case is $m/2$ (obtained with Booth's recoding). But experiments suggest that $q_{\text{worst}} = o(m)$. [8,9]

5.2 Average Case

Now we wish to obtain similar results for the average case. Again, let us consider the set O_m of the 2^{m-2} nonnegative odd integers having exactly m bits in their binary representation. As all the corresponding encoded programs must be different, the average code word length is at least:

$$\frac{1}{2^{m-2}} \sum_{i=1}^{2^{m-2}} \lfloor \log_2 i \rfloor = \frac{L(2^{m-2})}{2^{m-2}} = m - 4 + \frac{m}{2^{m-2}},$$

(where the function L was defined in Section 4.3), i.e.

$$\frac{1}{2^{m-2}} \sum_{i \in O_m} B(m, q_i) \geq m - 4 + \frac{m}{2^{m-2}}.$$

Therefore

$$2 + (2 + C(S(m)) + 2\,C(m)) \frac{1}{2^{m-2}} \sum_{i \in O_m} q_i \geq m - 4 + \frac{m}{2^{m-2}}$$

and

$$q_{\text{av}} \geq \frac{m - 6 + m/2^{m-2}}{2 + C(S(m)) + 2\,C(m)}.$$

Asymptotically, under the $S(m) = \alpha.m$ assumption, we obtain Theorem 5, i.e. the same bound as with the worst case.

Theorem 5 *Let $m \geq 2$. For any positive odd integer n having exactly m bits in its binary representation, consider an acceptable sequence computing n and let q_n denote its length. Let $q_{\text{av}} = 2^{2-m} \sum_n q_n$ be the average length. Assume that the shift counts are bounded above by a function $S(m) = \alpha.m$ (α being a positive constant). Then*

$$q_{\text{av}} \gtrsim \frac{m}{3\,\log_2(m)}.$$

Again, we do not know if the lower bound given by Theorem 5 is reached. The only currently known upper bound for the average case is $m/3$ (obtained with Booth's recoding).

5.3 The Case of Bernstein's Algorithm

With Bernstein's algorithm (described in [3,4,8,9]), an elementary operation can only be one amongst:

$$u_i = \begin{cases} 2^{c_i}\,u_{i-1} - 1 & \text{with } c_i \geq 1, \\ 2^{c_i}\,u_{i-1} + 1 & \text{with } c_i \geq 1, \\ (2^{c_i} - 1)\,u_{i-1} & \text{with } c_i \geq 2, \\ (2^{c_i} + 1)\,u_{i-1} & \text{with } c_i \geq 2, \end{cases}$$

and we know that the shift count c_i (for any i) is always bounded above by $S(m) = m$. Contrary to the generic elementary operation of our formulation, only one integer (i.e. c_i) needs to be encoded per operation, instead of 3. As a consequence, in the asymptotic lower bounds, instead of having a factor 3, we have a factor 1 (and these bounds are proved, as we do not need any assumption on the function S):

$$q_{\text{worst}} \gtrsim \frac{m}{\log_2(m)} \quad \text{and} \quad q_{\text{av}} \gtrsim \frac{m}{\log_2(m)}.$$

It is probably possible to find larger lower bounds using the fact that the sum of the shift counts has the same magnitude as m.

6 Conclusion and Acknowledgements

We gave lower bounds on the length of code that performs a multiplication by a constant, according to a given formulation. Such bounds are not completely proved; so, future work could consist in completing the proof (perhaps by developing techniques introduced in this paper), but also in improving these bounds.

The results given in Section 3.3 required several weeks of computations; these were performed on a machine from the MEDICIS [4] computation center; I wish to thank them.

References

[1] R. Beigel. Unbounded searching algorithms. *SIAM Journal on Computing*, 19(3):522–537, 1990.

[2] J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, August 1976.

[3] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[4] P. Briggs and T. Harvey. Multiplication by integer constants. ftp://ftp.cs.rice.edu/public/preston/optimizer/multiply.ps.gz, July 1994.

[5] R. Fredrickson. Constant coefficient multiplication. Master's thesis, Brigham Young University, December 2001.

[6] D. Knuth. Supernatural numbers. In D. A. Klarner, editor, *The Mathematical Gardner*, pages 310–325. Wadsworth International, Belmont, CA, 1981.

[7] V. Lefèvre. Multiplication by an integer constant. Research report RR1999-06, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

[8] V. Lefèvre. Multiplication by an integer constant. Research report RR-4192, INRIA, May 2001.

[9] V. Lefèvre. Multiplication par une constante. *Réseaux et Systèmes Répartis, Calculateurs Parallèles*, 13(4–5):465–484, 2001.

[10] J.-C. Raoult and J. Vuillemin. Optimal unbounded search strategies. Research report 33, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, 1979.

[4] http://www.medicis.polytechnique.fr/